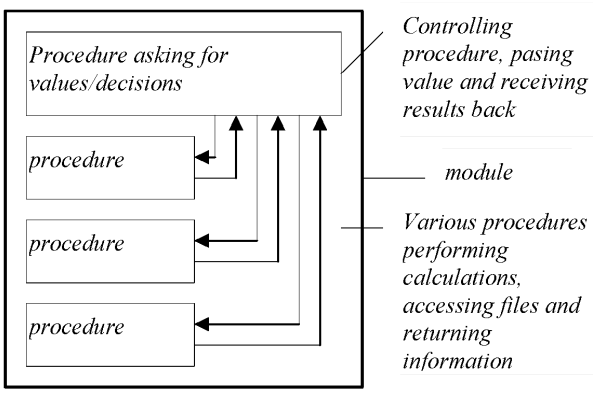
4 Calling procedures

USING MORE THAN ONE PROCEDURE

If you wanted a single procedure to perform a complex task, the procedure would become long and complicated. It is more convenient to have a module containing a number of procedures, each of which you can write and edit separately.

Many OPL modules are in fact a set of procedures linked up - each procedure doing just one job (such as a certain calculation) and then passing its results on to other procedures, so they can do other operations:



OPL is designed to encourage programs written in this way, since:

- You can store all the procedures which make up a program in the same module file

and

- One procedure can *call*, that is run, another.

Modules containing more than one procedure

You can have as many procedures as you like in a module. Each must begin with PROC and end with ENDP.

When you run a translated module it is always the first procedure, at the top of the module, which is actually run. When this finishes, the module stops; any other procedures in the file are only run if and when they are called.

Although you can use any name you want, it's common to give the first procedure a name like `start`.

Procedures which run on their own should be written and translated as separate modules, otherwise you won't be able to run them.

Calling procedures

To run another procedure, simply give the name of the procedure (with the colon). For example, this module contains two procedures:

```
PROC one:
  PRINT "Start"
  PAUSE 40
  two:                          REM calls procedure two:
  PRINT "Finished"
  PAUSE 40
ENDP
```

```
PROC two:
  PRINT "Doing..."
  PAUSE 40
ENDP
```

Running this module would run procedure `one:`, with this effect: `Start` is displayed; after a PAUSE it calls `two:`, which displays `Doing...`; after another PAUSE `two:` returns to the `one:` procedure; `one:` displays `Finished`; and after a final PAUSE, `one:` finishes.

Î Remember the 'Go to' button on the toolbar allows you to jump between procedures, for quick navigation around the module.

Ì Remember the ◆ key allows you to switch between a 'Normal' and an 'Outline' view of your OPL module. The 'Outline' view lists only the names of each procedure, for quick navigation around the module.

Uses of calling procedures

Calling procedures can be used to:

- Structure your programs more clearly so they're easier to adapt after you've written them
- Use the same procedure in different programs - say, to perform a certain common calculation.

For example, when your program asks you "Do this or do that?", make two procedure calls - either `this:` or `that:` procedure - depending on what you reply, for example:

```
PROC input:
  LOCAL a$(1)
  PRINT "Add [A] or Subtract [S]?:",
  a$=UPPER$(GET$)
  IF a$="A"
    add:                                REM first procedure
  ELSEIF a$="S"
    subtract:                           REM second procedure
  ENDIF
ENDP
```

To make full use of procedure calls, you must be able to communicate values between one procedure and another. There are two ways of doing this: *global variables* and *parameters*.

PARAMETERS

Values can be passed from one procedure to another by using *parameters*. They look and act very much like arguments to functions.

In the example below, the procedure `price:` calls the procedure `tax:`. At the same time as it calls it, it passes a value (in this case, the value which INPUT gave to the variable `x`) to the parameter `p` named in the first line of `tax:`. The parameter `p` is rather like a new local variable inside `tax:`, and it has the value passed when `tax:` is called. (The `tax:` procedure is **not** changing the variable `x`.)

The `tax:` procedure displays the value of `x` plus 17.5% tax.

```
PROC price:
   LOCAL x
   PRINT "ENTER PRICE",
   INPUT x
   tax:(x)            REM Passes the value of x to p
   GET
ENDP

PROC tax:(p)
   PRINT "PRICE INCLUDING TAX =",p*1.175
ENDP
```

- In the *called* procedure, follow the procedure name by the names to be used for the parameters, enclosed by brackets and separated by commas - for example `proc2:(cost,profit)`.

The parameter type is specified as with variables - for example `p` for a floating-point parameter, `p%` for an integer, `p&` for a long integer, `p$` for a string. You can't have array parameters.

- In the *calling* procedure, the *values* for the parameters are given in brackets, in the right order and separated by commas, after the colon of the called procedure - for example `proc2:(60,30)`.

The values passed as parameters may be the values of variables, strings in quotes, or constants. So a call might be `calc:(a$,x%,15.8)` and the first line of the called procedure `PROC calc:(name$,age%,salary)`

In the called procedure, you cannot assign values to parameters - for example, if `p` is a parameter, you cannot use a statement like `p=10`.

You will see a 'Type mismatch' error displayed if you try to pass the wrong type of value to a parameter - for example, `45` to `(a$)`.

Multiple parameters

In the following example, the second procedure `tax2:` has two parameters:

- The value of the price variable `x` is passed to the parameter `p1`.
- The value of the tax rate variable `r` is passed to the parameter `p2`.

`tax2:` displays the price plus tax at the rate specified.

```
PROC price2:
   LOCAL x,r
   PRINT "ENTER PRICE",
   INPUT x
   PRINT "ENTER TAX RATE",
   INPUT r
   tax2:(x,r)
   GET
ENDP

PROC tax2:(p1,p2)
   PRINT p1+p2 %
ENDP
```

This uses the % symbol as an operator - p1+p2 % means p1 plus p2 percent of p1. Note the space before the %; without it, p2% would be taken as representing an integer variable.

Appendix B has more about the % operator.

Returning values

In the following example, the RETURN command is used to return the value of x plus tax at r percent, to be displayed in price3:. This is very similar to the way functions return a value.

The tax3: procedure calculates, but doesn't display the result. This means it can be called by other procedures which need to perform this calculation but do not necessarily need to display it.

```
PROC price3:
   LOCAL x,r
   PRINT "ENTER PRICE",
   INPUT x
   PRINT "ENTER TAX RATE",
   INPUT r
   PRINT "PRICE INCLUDING TAX =",tax3:(x,r)
   GET
ENDP

PROC tax3:(p1,p2)
   RETURN p1+p2 %

ENDP
```

Only one value may be returned by the RETURN command.

The name of a procedure which returns a value must end with the correct identifier - $ for string, % for integer, or & for long integer. To return a floating-point number, it should end with none of these symbols. For example, PROC abcd$: can return a string, while PROC counter%: can return an integer. In this example, ref$: returns a string:

```
PROC refname:
   LOCAL a$(30),b$(2)
   PRINT "Enter reference and name:",
   INPUT a$
   b$=ref$:(a$)
   PRINT "Ref is:",b$
   GET
ENDP


PROC ref$:(name$)
   RETURN LEFT$(name$,2)
   REM LEFT$ takes first 2 letters of name$
ENDP
```

If you don't use the RETURN command, a string procedure returns the null string ("""). Other (numeric) types of procedure return zero.

GLOBAL VARIABLES

You can only return one value with the RETURN command. If you need to pass back more than one value, use *global* variables.

Instead of declaring `LOCAL x%,name$(5)` declare `GLOBAL x%,name$(5)`. The difference is that:

- Local variables are valid only in the procedure in which they are declared.
- Global variables can also be used in any procedures (including those in loaded modules) called by the procedure in which they are declared.

So this module would run OK:

```
PROC one:
  GLOBAL a%
  PRINT a%
  two:
  GET
ENDP

PROC two:
  a%=2                          REM Sees a% declared in one:
  PRINT a%
ENDP
```

When you run this, the value 0 is displayed first, and then the value 2.

You would see an 'Undefined externals' error displayed if you used LOCAL instead of GLOBAL to declare `a%`, since the procedure `two:` wouldn't recognise the variable `a%`. In general, though, it is good practice to use the LOCAL command unless you really need to use GLOBAL.

A local declaration overrides a global declaration in that procedure. So if `GLOBAL a%` was declared in a procedure, which called another procedure in which `LOCAL a%` was declared, any modifications to the value of `a%` in this procedure would not effect the value of the global variable `a%`.

Passing back values

You can effectively pass as many values as you like back from one procedure to another by using global variables. **Any modifications to the value of a variable in a called procedure are automatically registered in the calling procedure**. For example:

```
PROC start:
  GLOBAL varone,vartwo
  varone=2.5
  vartwo=2
  op:
  PRINT varone,vartwo
  GET
ENDP
```

```
PROC op:
   varone=varone*2
   vartwo=vartwo*4
ENDP
```

This would display `5 8`

‘Undefined externals’ error

If, perhaps because of a typing error, you use a name which is not one of your variables, no error occurs when you translate the module. This is because it could be the name of a global variable, declared in a different procedure, which might be available when the procedure in question was called. If no such global variable is available, an ‘Undefined externals’ error is shown when the translated module is run. This also displays the variable name which caused the error, together with the module and procedure names, in this format: ‘Error in *MODULE\PROCEDURE*, *VARIABLE*’.

SERIES 5 HEADER FILES, CONSTANTS AND PROCEDURE PROTOTYPES

On the Series 5, OPL allows you to *include header files* which may include definitions of *procedure prototypes* and *constants*, but **not** procedures themselves. (Constants and procedure prototypes may also be declared at the top of modules themselves, although it is tidier to put them into a header file. Indeed, including a file is logically identical to replacing the INCLUDE statement by the file’s contents.)

A header file is included in a module using the INCLUDE command at the beginning of the module, outside any procedure. For example,

```
INCLUDE "Header.oph"
```

The filename of the header may or may not include a path. If it does include a path, then OPL will only scan the specified folder for the file. However, the default path for INCLUDE is `\System\Opl\`, so when INCLUDE is called *without specifying a path*, OPL looks for the file firstly in the current folder and then in `\System\Opl\` in all drives from Y: to A: and then in Z:, excluding any remote drives.

Commonly the statement,

```
DECLARE EXTERNAL
```

will follow the INCLUDE declaration. DECLARE EXTERNAL causes the **translator** to report ‘Undefined externals’ errors if any variables or procedures are used before they are declared, rather than leaving this until runtime.

Procedure prototypes are declared with the command EXTERNAL. For example,

```
EXTERNAL Proc1:
```

A prototype is a declaration of the name of the procedure along with the arguments it takes. This amounts to the same as PROC declaration with the PROC keyword, which declares the start of a procedure, omitted. The procedure may then be referred to before it is defined when the DECLARE EXTERNAL statement has been made. As well as reporting ‘Undefined externals’ error at translate-time, the other advantage of using the DECLARE EXTERNAL and EXTERNAL statements is that it allows parameter type-checking to be performed at translate-time rather than at runtime, and also provides the necessary information for the translator to *coerce* numeric argument types, thus avoiding ‘Type violation’ errors at runtime. Hence a ‘Type violation’ error does not result in the following example, even though a `&` does not precede the 2 passed to the procedure `two:()`,

```
DECLARE EXTERNAL
EXTERNAL two:(long&)
PROC one:
   two:(2)
ENDP
```

```
PROC two:(long&)
  ..
ENDP
```

The same *coercion* occurs as when calling the built-in keywords.

Constants are declared with the command CONST. For example,

```
CONST KConstant=1.0
```

Constants are treated as literals, not stored as data. They also have *global scope* and once a value is assigned to them, it cannot be altered within the same program. The declarations must be made **outside** any procedure. A constant's name, just like that of a GLOBAL or LOCAL variable, has the normal type-specification indicators (`%`, `&`, `$` or nothing for floats). By convention, all constants are named with a leading `K` to distinguish them from variables.

Const.oph is the standard header file in the ROM. It provides many of the standard constant declarations required for effective and maintainable OPL programming on the Series 5. For convenient reference, the contents Const.oph is reproduced in full in Appendix E. This and other files stored in the ROM (for example, OPX header files: see the 'Using OPXs on the Series 5' chapter) may be created in RAM by using the 'Create standard files' option in the 'Tools' menu in the Program editor.

See also the 'Alphabetic Listing' chapter.

SUMMARY

Call a procedure by stating its name, including the colon.

Pass *parameters* to a procedure by following the procedure call with the values for the parameters, e.g. `calc2:(4.5,32)`. In the called procedure, follow the procedure name with the parameter names, e.g. `PROC calc2:(mod,div%)`.

To make variables declared in one procedure accessible to called procedures, declare the variables with GLOBAL instead of LOCAL.

Î INCLUDE may be used to *include* a *header file* which contains constant definitions and procedure prototypes.

DECLARE EXTERNAL may be used to

- cause the translator to report an error if any variables or procedures are used before they are declared
- allow parameter type-checking to be performed at translate-time rather than at runtime
- provide the necessary information for the translator to coerce numeric argument types.

Procedure prototypes are made with the EXTERNAL command.

Constant definitions are made with the CONST command.